Register Allocation using Bipartite Liveness Graphs

Rajkishore Barik, Intel Corporation, Santa Clara, California
Jisheng Zhao, Rice University, Houston, Texas
Vivek Sarkar, Rice University, Houston, Texas

Technical Report TR 10-10
Department of Computer Science
Rice University
November 2010

Abstract

Register allocation is an essential optimization for all compilers. A number of sophisticated register allocation algorithms have been developed based on Graph Coloring (GC) over the years. However, these algorithms pose three major limitations in practice. First, construction of a full interference graph can be a major source of space and time overhead in the compiler. Second, the interference graph lacks information on the program points at which two variables may interfere. Third, integration of coloring and coalescing leads to a coupling between register allocation and register assignment, which can further compromise the effectiveness of the solution. This paper addresses these limitations by making a clean separation between the register allocation and register assignment phases. Allocation is modeled as an optimization problem on a new data structure called the Bipartite Liveness Graph (BLG). We model the register assignment phase as a separate optimization problem that avoids some spill instructions by generating register-to-register moves and exchange instructions and at the same time, performs move coalescing and handles register class constraints.

We implemented our BLG allocator in both the LLVM static compiler infrastructure and the Jikes RVM dynamic compiler infrastructure. In the LLVM evaluation, our BLG register allocator results in a performance improvement of up to 7.8% for SpecCPU 2006 benchmarks and a significantly lower compile-time overhead compared to a Chaitin-Briggs GC register allocator with both allocators using the same spill-code generator. The BLG allocator delivers performance comparable to the existing LLVM 2.7 Linear Scan register allocator that includes additional optimizations such as live-range splitting and backtracking techniques that are currently not present in the BLG allocator. In the Jikes RVM evaluation, the BLG register allocator delivers runtime performance improvements of up to 30.7% for Java Grande Forum benchmarks and up to 9% for Dacapo benchmarks relative to Linear Scan (LS) with a modest increase in compile-time.

1. Introduction

Register allocation is an essential compiler optimization that has received much attention from the research community during the last five decades. Its relevance continues to increase with current trends towards energy-efficient processors in which some of the burden of memory hierarchy management is shifting back from hardware to software. Three key metrics for the quality of a register allocator are *compile-time*, *compile-space*, and *execution time*. Past work has explored different trade-offs across these metrics in many different ways.

The primary goal of this paper is to explore algorithms with polynomial compile-time and linear compile-space that deliver the best execution time performance possible. The motivation for polynomially bounded compile-times is that large compile-times can have a major impact on overall programmer productivity in scenarios such as programmer-directed performance tuning and automatic adaptive and dynamic optimization. This rules out the use of register allocation algorithms with worst-case exponential compile-times such as [Appel and George 2001, Hames and Scholz 2006, Grund and Hack 2007] to achieve our goal. The motivation for the linear compile-space constraint is that memory is a critical resource for all applications, including compilers, and that non-linear space leads to the creation of large data structures that don't fit in lower levels of cache when compiling large procedures, thereby further contributing to compile-time increases. Register allocation algorithms based on Graph Coloring (GC) [Chaitin et al. 1981, Briggs et al. 1994, George and Appel 1996, Park and Moon 1998, Smith et al. 2004], including more recent variants based on Static Single Assignment (SSA) form [Hack and Goos 2006, Pereira and Palsberg 2009] all use the *Interference Graph (IG)* as a primary data structure which is often super-linear in size. Register allocation algorithms based on Linear Scan (LS) *e.g.,* [Traub et al. 1998, Poletto and Sarkar 1999, Wimmer and Mössenböck 2005, Sarkar and Barik 2007, Wimmer and Franz 2010] overcome the compile-time and compile-space overheads of GC algorithms, but do so at the expense of achieving poorer execution times than GC.

A secondary goal of this paper is to simplify the implementation of the register allocator by decoupling the *register allocation* and *register assignment* phases in an optimizing back-end. This will allow the allocation phase to focus on spilling decisions and the assignment phase can focus on coalescing and physical register assignment decisions. While this form of decoupling has been performed for other register allocation algorithms in the past [Appel and George 2001], our approach is unique in its use of the Bipartite Liveness Graph (BLG) for the allocation phase and the Coalesce Graph (CG) for the assignment phase. The CG consists of both IR move instructions and register-to-register moves that arise from our BLG based allocation phase. In GC algorithms, the coupling between these phases is manifest in the integration of coloring and coalescing decisions, which can further compromise the effectiveness of the final solution and complicate the implementation of the allocator. These complications arise from non-trivial problems that must be addressed by the implementer in dealing with coalescing in traditional GC allocators and with optimization of ϕ-function copy statements in SSA-based GC allocators. Further, register allocation for today's architectures includes new challenges due to hardware features such as *register classes*, *register aliases*, *pre-coloring*, and *register pairs*. To produce high quality machine code, a register allocator must consider these hardware features in both the allocation and assignment phases.

This paper addresses these challenges by starting with a clean separation between the register allocation and register assignment phases. Allocation is modeled as an optimization problem on a new data structure called the *Bipartite Liveness Graph (BLG)*. As we will see, the BLG is a more compact data structure than the IG, and it achieves linear compile-space in practice even though its worst-case compile-space is quadratic[1]. Assignment is modeled as a separate optimization problem that incorporates register-to-register moves and exchanges as alternatives to spilling, and handles move coalescing and register class constraints.

Specifically, we make the following contributions towards the above goals:

1. We introduce a novel *Bipartite Liveness Graph* (*BLG*) representation as an alternative to the interference graph (*IG*) representation. Interestingly, the *BLG* is both more compact and more precise than the *IG* in practice.
2. We formulate the *allocation* problem for BLGs as a simple optimization problem and present a greedy heuristic to solve it. The allocation phase is performed independently of coalescing optimizations.
3. We formulate *spill-free register assignment with move coalescing* as a combined optimization problem that maximizes the benefits of move coalescing while finding an assignment for every symbolic register. Move coalescing is performed on a *Coalesce Graph* (*CG*). A local greedy heuristic is presented to address the assignment optimization problem.
4. We extend the register assignment approach from 3. above to handle *register classes*. An optimized version of the assignment problem is presented that *minimizes the additional spilled symbolic registers and, at the same, time maximizes the benefits of move coalescing*. A prioritized bucket based greedy heuristic is presented to address this problem.
5. We present *experimental results* for implementations of BLG register allocation in both the LLVM static compiler infrastructure for C programs and the Jikes RVM dynamic compiler infrastructure for Java programs. For the LLVM comparison, we use ten SpecCPU 2006 benchmarks and compare our register allocator with that of an LLVM implementation of a Chaitin-Briggs GC register allocator. Our *BLG* register allocator results in a performance improvement of up to 7.87%, while incurring a significantly lower compile-time overhead than GC. We use the serial version of the Java Grande benchmark suite and Dacapo benchmark suite to compare our *BLG* based register allocation with that of existing Linear Scan (*LS*) register allocation in Jikes RVM. The results show that a *BLG* based register allocation can achieve runtime benefits of up to 30.7% for Java Grande and of up to 9% for Dacapo compared to *LS*.

2. Bipartite Liveness Graph (BLG)

A program point can be split into two program points based on the values read and written at that program point [Sarkar and Barik 2007]:

DEFINITION 2.1. *Each program point p is split into p^- and p^+, where p^- consists of the variables that are read at p and p^+ consists of the variables that are written at p.*

$[x, y]$ is called a basic interval for variable v (denoted as $\texttt{BI}(v)$) if and only if for every program point, p, such that $p \geq x$ and $p \leq y$ imply v is live at p. Note that $\texttt{BI}(v)$ does not include any hole. x and y denote the start and end points of $\texttt{BI}(v)$ respectively. A compound interval for a variable v (denoted as $\texttt{CI}(v)$) consists of a set of basic intervals for v. $\texttt{CI}(v)$ can have holes. Let $\mathcal{B}$ denote the set of all basic intervals and $\mathcal{C}$ denote the set of all compound intervals in the program. Let $\mathcal{L}$ denote the set of start points and $\mathcal{H}$ denote the set of end points of all the basic intervals.

The number of simultaneously live symbolic registers at a program point p is denoted by *numlive*(p). *MAXLIVE* represents the maximum number of simultaneously live symbolic registers in any program point. A program point p is said to be *constrained* if *numlive*$(p) > k$, where k is the total number of machine registers. In the presence of register classes, we call a program point p *constrained* if it violates any of the register requirements of any of the register classes of the symbolic registers that are live at p.

Now we present a new representation, known as *Bipartite Liveness Graph* (*BLG*), that captures program point specific liveness information as an alternative to the interference graph. Formally,

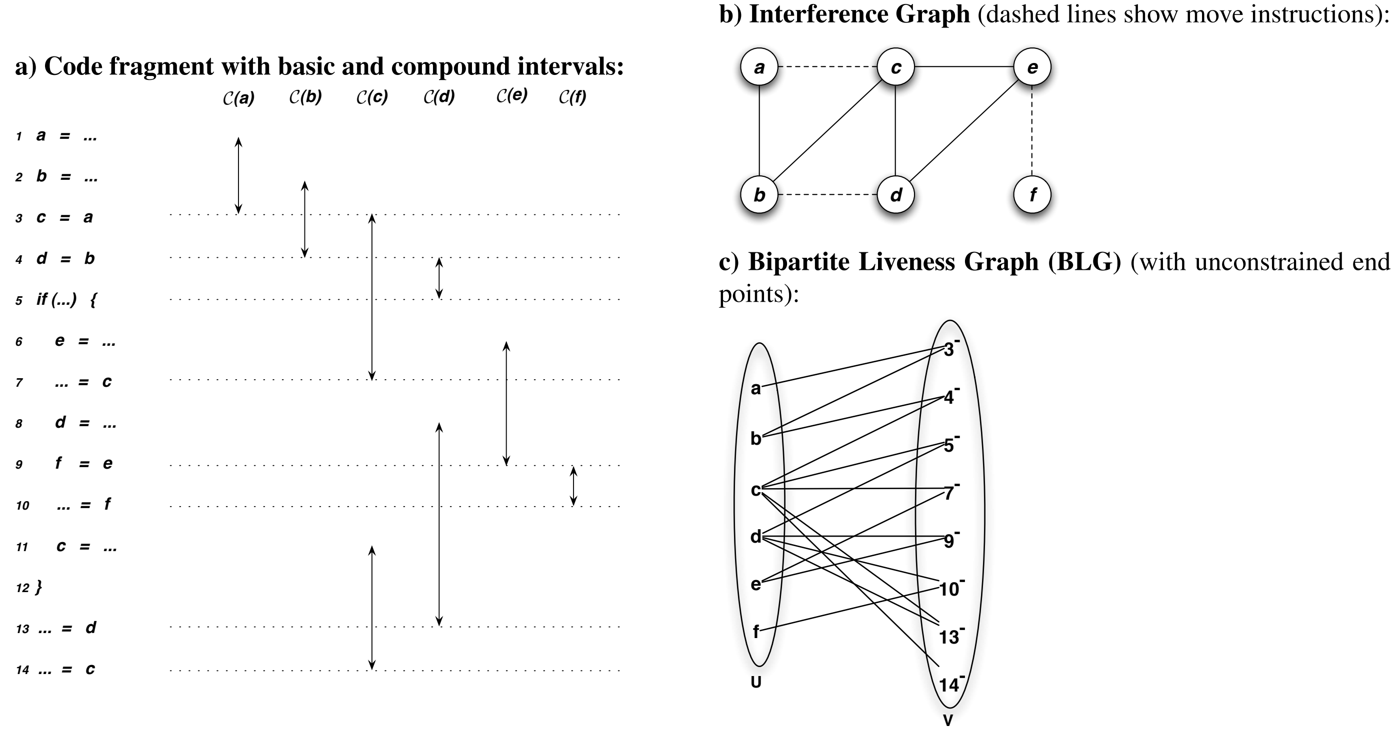
DEFINITION 2.2. **Bipartite Liveness Graph:** *A bipartite liveness graph (*BLG*) is a undirected weighted bipartite*[2] *graph $G = \langle U \cup V, E\rangle$, where V denotes all the basic interval end points*[3] *in $\mathcal{H}$, U denotes all the compound intervals in $\mathcal{C}$ and an edge $e = (u, v) \in E$ indicates that the compound interval $u \in U$ is* live *at the interval end point $v \in V$. Each $u \in U$ has an associated non-negative weight SPILL(u) that denotes the spill cost of u. Similarly, each $v \in V$ has an associated non-negative weight FREQ(v) that denotes the execution frequency of the* IR *instruction associated with basic interval end point v.*

---

[1] This is analogous to SSA form, which has worst-case quadratic compile-space, but is observed to exhibit linear compile-space in practice.

[2] A bipartite graph is a graph whose vertices can be divided into two disjoint sets U and V such that each edge connects a vertex in U to one in V.

[3] The choice of interval end points is arbitrary. We could have used interval start points instead.



a) Code fragment with basic and compound intervals:

```
1  a = ...
2  b = ...
3  c = a
4  d = b
5  if (...) {
6     e = ...
7     ... = c
8     d = ...
9     f = e
10    ... = f
11    c = ...
12 }
13 ... = d
14 ... = c
```

b) Interference Graph (dashed lines show move instructions):

c) Bipartite Liveness Graph (BLG) (with unconstrained end points):

Figure 1. a) Example code fragment with basic and compound intervals; the dotted lines represent end-points of basic intervals. b) Interference Graph (*IG*); the solid lines in *IG* represent interference and the dashed lines represent move instructions. c) Bipartite Liveness Graph (*BLG*) with unconstrained interval end-points; the vertices on the left of the graph represent compound intervals, and the vertices on the right represent basic interval end-points. With two physical registers, the *BLG* representing constrained end-points is empty in this case.

It is obviously a waste of space to capture liveness information at every program point in V of *BLG*. From a register allocation perspective, it suffices to consider only *constrained* program points corresponding to either the basic interval start points alone or end points alone but not both in V. This is because spilling/assignment decisions only need to be taken at those points. Additional optimizations are also possible, *e.g.,* if two interval end points have the same liveness information (*i.e.,* same set of variables live), only one of them (but not both) needs to be added to the *BLG* for spilling decisions.

Figure 1 presents an example code fragment with its basic and compound intervals in Figure 1a) and the interference graph (*IG*) in Figure 1b). We observe that *IG* has a clique of size 3 due to the cycle comprising nodes c, d, and e. Now consider a Graph Coloring register allocator that performs coalescing along with register allocation. Both aggressive [Chaitin et al. 1981] and conservative [Briggs et al. 1994] coalescing will be able to eliminate the move edges (a, c), (b, d), and (e, f) without increasing the colorability of the original interference graph. If we have two physical registers, we have to spill one of the coalesced nodes ac, bd, and ef. The un-coalescing approach used in an optimistic coalescing technique [Park and Moon 1998] will be able to just spill one of the nodes involved in the cycle as it tries all possible combinations of assigning colors to individual nodes of a potentially spilled coalesced node. The points to note here are that we can not color the *IG* using 2 physical registers and that opportunities for coalescing can be missed due to the inability to color certain nodes.

A closer look at the code reveals the fact that none of the program points have more than two variables live simultaneously. If this is the case, two questions come to mind: 1) Can we generate spill-free code with two physical registers that does not give up any coalescing of symbolic registers? 2) If the answer to the first question is yes, then why did Graph Coloring generate spill code and also miss the coalescing opportunity?

The answer to the first question is yes. The *BLG* with unconstrained interval end points for the example code is shown in Figure 1(c). This captures the fact that every basic interval end point in V has degree less than or equal to 2 indicating no more than two compound intervals are simultaneously live. (The *BLG* with contrained interval end points is empty in this case.) Let us name the two physical registers as r_1 and r_2. The following register assignment is possible: $reg([1^+, 3^-]) = r_1$, $reg([2^+, 4^-]) = r_2$, $reg([4^+, 5^-]) = r_2$, $reg([3^+, 7^-]) = r_1$, $reg([6^+, 9^-]) = r_2$, $reg([9^+, 10^-]) = r_2$, $reg([8^+, 13^-]) = r_1$, and $reg([11^+, 14^-]) = r_2$. This register assignment requires an additional register exchange operation since the register assignment for the basic intervals of both `CI`(c) and `CI`(d) were exchanged when the code after the `if` condition was executed.

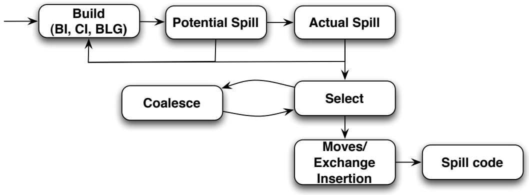


Figure 2. Register Allocation using BLG

We need to insert an $exchg\ r_1, r_2$ instruction on the control flow edge between 4 and 13. As a result none of the coalescing opportunities in lines 3, 4, and 9 were given up during such an assignment.

Now let us try to answer the second question. Looking at the code fragment, we observe that at the program point 13^-, d interferes with two values of c that are assigned on lines 3 and 11. Similarly, c interferes with two values of d that are assigned on lines 4 and 8. During runtime, if the `if` branch is taken then assignments on lines 8 and 11 will be visible to the code following the `if` condition, otherwise assignments on lines 3 and 4 will be visible. This notion can not be precisely captured using the definition of *live-ranges* in an interference graph unless we convert the program to *SSA* form or perform live-range splitting [Appel and George 2001]. Each of these approaches require additional complexities, *e.g.,* the *SSA*-based approach needs to handle out-of-SSA translation by inserting extra copy statements.

The above example raises a question about the general approach of stating the global register allocation problem as the graph coloring problem on the *IG*. Even though the interference graph using live-ranges provides a global view of the program, it is less precise than a *BLG* with intervals. Additionally, the interference graph (in terms of size and building it) is known to be a major compile-time bottleneck [Cooper and Dasgupta 2006, Sarkar and Barik 2007].

3. Overall Approach

The overall register allocator presented in this paper is depicted in Figure 2. The first step in the allocator is to build data structures for basic intervals, compound intervals, and the Bipartite Liveness Graph (*BLG*). Then, the *allocation* is performed on the *BLG* to determine a set of compound intervals that need to be spilled everywhere as shown in the blocks for *potential spill* and *actual spill*. A combined phase of assignment and coalescing is then performed until all the symbolic registers are assigned physical registers or spilled. Next, register move and exchange instructions are added to the *IR* to produce correct code. Finally, spill code is added to the IR. Our approach of separating register allocation and register assignment phases has also been done in past work [Appel and George 2001].

4. Allocation using Bipartite Liveness Graphs

As in Linear Scan and other simple register allocation algorithms, we take an all-or-nothing approach for spills in this paper: if a symbolic register is selected for spilling, every access of the symbolic register in the program will be replaced by a load or store instruction.[4]

DEFINITION 4.1. **Allocation Optimization Problem:** *Given a* BLG *with constrained end-points, G, and k uniform physical registers, find a spill set $S \subseteq U$ and $G' \subseteq G$ induced by S such that: (1) $\forall v \in V$, v is unconstrained,* i.e., *DEGREE$(v) \leq k$; and (2) $\sum_{s \in S}$ SPILL(s) is minimized. For each compound interval $s \in S$ and basic interval $b \in s$, set $spilled(b) := true$.*

Given a *BLG*, the register allocation problem now reduces to an optimization problem whose solution ensures that no more than k physical registers are needed at every interval end point, and at the same time, spills as few compound intervals as possible. Algorithm 3 provides a greedy heuristic that solves the allocation optimization problem. Steps 3-11 choose *Potential Spill* candidates (as shown in Figure 2) using a *max-min* heuristic. Each iteration of the loop alternates between largest frequency interval end point and smallest spill cost symbolic register. The alternating approach allows the option of completely unconstraining a high pressure region of program points before moving onto another. Steps 12-15 *unspill* some of the potential spill candidates resulting *Actual Spill* (as shown in Figure 2) candidates. The unspilling step reverts a potential spill candidate and its edges back onto the *BLG* and verifies if the *BLG* becomes constrained after adding the potential spill candidate. If the *BLG* does not get constrained, then the symbolic register can be unspilled. Depending on the quality of potential spill candidate selection, the unspilling of spill candidates provides a way of rectifying the obvious spilling mistakes (akin to *unspilling* in Graph Coloring).

THEOREM 4.1. *Algorithm 3 ensures that every program point has k or fewer number of symbolic registers simultaneously live.*

---

[4] Extending the *BLG* approach to partial spills is a topic for future research.

```
1  function GreedyAlloc()
     Input  : Weighted Bipartite Liveness Graph $G = \langle U \cup V, E \rangle$ and $k$ uniform physical registers
     Output: Set $T \subseteq U$ which needs to be spilled to ensure all interval end points $v \in V$ be unconstrained i.e., $\forall b \in T$, $spilled(b) = true$
2    Stack $S := \phi$;
     //Potential spill selection
3    $n$ := Choose a constrained node $n \in V$ with largest FREQ($n$);
4    while $n$ != null do
5        $s$ := Choose a compound interval $s \in U$ having an edge to $n$ and has smallest SPILL($s$);
6        Push $s$ on to $S$; Delete edge $(s, n)$;
7        $n$ := Choose a constrained node $n \in V$ having an edge to $s$ and has largest FREQ($n$);
8        if $n$ == null then
9            $n$ := Choose a constrained node $n \in V$ with largest FREQ($n$);
10       Delete all edges incident on $s$;
11       Remove $s$ from $G$;
     //Actual spill selection
12   while $S$ is not empty do
13       $s := pop(S)$;
14       if $\forall n \in V$, $n$ becomes constrained by reverting $s$ and its edges in $G$ then
15           $spilled(s) := true$; $T := T \cup \{s\}$;
16   return $T$
```

Figure 3. Greedy heuristic to perform allocation

Proof: This is trivial as the algorithm continues to execute the while loop in Steps 4-11 until there are constrained nodes $v \in V$ in the BLG. This is guaranteed by steps 3, 7, and 9. □

THEOREM 4.2. *Given the bipartite liveness graph, the Algorithm 3 requires* $\mathcal{O}(|\mathcal{H}| * max(0, (MAXLIVE - k)) * |\mathcal{C}|)$ *time.*

Proof: Every interval end point in $\mathcal{H}$ is traversed at most *MAXLIVE* $- k$ number of times to make it unconstrained. To make an interval end point unconstrained, we need to visit all its neighbor and choose a minimum spill cost compound interval. This requires, at most, $|\mathcal{C}|$ edge visits. □

One of the advantages of Algorithm 3 is that if a spill-free allocation exists, the algorithm is guaranteed to find an allocation without spills. On the other hand, if one works with an allocator based on graph coloring, it is an NP-hard problem to determine if a spill-free allocation exists. This seeming contradiction arises because *BLG* may require the insertion of register-copy instructions (described in Section 5), whereas the standard graph coloring algorithm does not allow for this possibility. Prior work on SSA-based register allocation [Hack and Goos 2006, Brisk et al. 2005, Bouchez 2009] and on Extended Linear Scan [Sarkar and Barik 2007] independently established that the existence of a spill-free allocation can be determined in polynomial time, provided that extra register-copy instructions can be inserted. In the case of SSA-based register allocation, the extra copies arise from ϕ-functions; in the case of Extended Linear Scan, they arise from the need to map from the register assignment for a symbolic register to another on a control flow edge. In both cases, the task of optimizing the additional copy instructions is a non-trivial problem.

5. Assignment using Register Moves and Exchanges

The allocation phase ensures that every program point needs k or fewer physical registers. In this section, we first describe how assignment for basic intervals can be performed by possibly adding extra register moves/exchanges to the *IR* without spilling any symbolic registers.

5.1 Spill-Free Assignment

DEFINITION 5.1. **Spill-free Assignment:** *Given a set of basic intervals* $b \in \mathcal{B}$ *with* $spilled(b) = false$*, and* k *uniform physical registers, find register assignment* $reg(b)$ *for every basic interval,* $b \in \mathcal{B}$*, including any register-to-register copy or exchange instructions that need to be inserted in the* IR.

The algorithm to perform register assignment for basic intervals is provided in Algorithm 4. The algorithm sorts the basic intervals in increasing start points. Steps 4-11 perform assignment to basic intervals using an $avail$ list of physical registers. The assignment to a basic interval first prefers getting the physical register that was previously assigned to another basic interval of the same compound interval (as shown in Step 7). This avoids the need for additional move/exchange instructions. However, in cases where the already assigned physical register is unavailable, we assign a new available physical register (as shown in Step 10). Assigning such a new physical register may produce incorrect code without additional move/exchange instructions on certain control flow paths.

Steps 12-20 of Algorithm 4 create a list of move instructions that need to be inserted on a control flow edge. These move instructions form the nodes of a directed anti-dependence graph D in Algorithm 5. The edges in D represent the anti-dependence between a pair of move instructions. Steps 5-10 of Algorithm 5 add the anti-dependence edges to D. A strongly connected component (SCC) search is performed on D to generate efficient code using *exchange* instructions for SCC's of size 2 or more (a shown in Steps 11-18). The nodes in a SCC are collapsed to a single node with exchange instructions. Finally, a topological sort order of D produces the correct code for a control flow edge e.

```
1  function RegMoveAssignment()
       Input  : IR, Set of basic intervals $b \in \mathcal{B}$ with $spilled(b) = false$ and $k$ uniform physical registers
       Output: $\forall b \in \mathcal{B}$, return the register assignment $reg(b)$ and any register moves and exchange instructions
2      $M := \phi$;
3      $avail$ := set of physical registers;
4      for each basic interval $b := [x, y]$, in increasing start points i.e., $\mathcal{L}$ do
5          for each basic interval $b' := [x', y']$ such that $y' < x$ do
6              $avail := avail \cup reg(b')$;
7          $r$ := find a physical register $p \in avail$ that was assigned to another basic interval of the same compound interval;
8          if $r$ == null then
9              Assert avail is not empty;
10             $r$ := find a physical register $p \in avail$;
11         $reg(b) := r$; $avail := avail - \{r\}$;
12     for each control flow edge, e do
13         for each compound interval $c \in \mathcal{C}$ that is live at both end points of $e$ do
14             $b_1$ := basic interval of $c$ at the source of $e$;
15             $b_2$ := basic interval of $c$ at the destination of $e$;
16             if $b_1$ != null and $b_2$ != null then
17                 $r_1 := reg(b_1)$; $r_2 := reg(b_2)$;
18                 if $r_1$ != $r_2$ then
19                     m := generate a new move instruction that moves $r_1$ to $r_2$ i.e., $mov\ \ r_2, r_1$;
20                     $M := M \cup \{m\}$;
21         GenerateMoves(IR, $M$,$e$);
22     return $T$ and IR
```

Figure 4. Assignment using register moves and exchange instructions

LEMMA 5.1. *The assertion on line 9 of Algorithm 4 never fails.*

Proof: Follows from the fact that every interval end point has no more than k symbolic registers simultaneously live.□

THEOREM 5.2. *Spill-free assignment takes $\mathcal{O}(|\mathcal{E}| * (|\mathcal{C}| + |\mathcal{K}|^2))$ space where $\mathcal{E}$ represents the control flow edges in a program and $\mathcal{K}$ represents the available physical registers.*

Proof: Additional space requirement in assignment phase is due to the anti-dependence graph D. For every control flow edge $e \in \mathcal{E}$, in the worst case we need to insert $|\mathcal{C}|$ number of register-to-register move instructions. These are the number of nodes in D. The number of edges in D are bounded by the square of physical registers $\mathcal{K}$, *i.e.,* it represents all possible anti-dependences between all possible pairs of physical registers. Hence the overall space complexity is $\mathcal{O}(|\mathcal{E}| * (|\mathcal{C}| + |\mathcal{K}|^2))$.□

THEOREM 5.3. *Spill-free assignment takes $\mathcal{O}(|\mathcal{B}| + (|\mathcal{E}| * (|\mathcal{C}| + |\mathcal{K}|^2)))$ time.*

Proof: Similar in nature to the proof for Theorem 5.2.□

5.2 Assignment with Move Coalescing and Register Moves

Move coalescing is an important optimization in register allocation algorithms that assigns the same physical registers to the source and destination of an *IR* move instruction when possible to do so. The register assignment phase must try to coalesce as many moves as possible so as to get rid of the move instructions from the *IR*. As we saw in the preceding section, additional register moves may be inserted in the assignment phase instead of spilling. Note that move coalescing approaches using

```
function GenerateMoves()
    Input : IR, Set of move instructions $M$ and a control flow edge $e$
    Output: Modified IR with register move and exchange instructions added
    $D := \phi$; //$D$ is the anti-dependence graph
    for $m_1 \in M$ do
        Add a node for $m_1$ in $D$;
    for $m_1 \in D$ do
        for $m_2 \in D$ and $m_2 != m_1$ do
            $s_1$ := source of the move instruction in $m_1$;
            $d_2$ := destination of the move instruction in $m_2$;
            if $s_1 == d_2$ then
                Add a a directed edge $(m_1, m_2)$ to $D$;
    $S$ := Find strongly connected components in $D$;
    for each $s \in S$ do
        Collapse all the nodes in $s$ to a single node $n$ in $D$;
        while number of move instructions in $s > 1$ do
            $m_1$ := Remove first move instruction from $s$;
            $m_2$ := First move instruction in $s$;
            $x$ := Generate an exchange instruction between the destinations of $m_1$ and $m_2$;
            Append $x$ to the instructions of $n$;
    for each node $n$ in $D$ in topological sort order do
        Add the move or exchange instructions of the node $n$ to the IR along the control flow edge $e$;
    return Modified IR
```

Figure 5. Insertion of move and exchange operations on a control flow edge

aggressive [Chaitin et al. 1981], conservative [Briggs et al. 1994], and optimistic [Park and Moon 1998] techniques are shown to be NP-complete by Bouchez et al [Bouchez et al. 2007]. In this section, we first present a *coalesce graph* that models both the *IR* move instructions and register-to-register moves. Then, the register assignment phase on the coalesce graph is formulated as an optimization problem that tries to maximize the number of move instructions removed after assignment. We provide a greedy heuristic to solve it.

DEFINITION 5.2. *A Coalesce Graph (*CG*) is an undirected weighted graph $G = \langle V, E_m \cup E_r \rangle$ where V represents the basic intervals in $\mathcal{B}$ and an edge $e \subseteq V \times V$ corresponds to the following two types of move instructions between a pair of basic intervals:*

1. E_m**:** *the move instructions already present in the* IR. *The weight of such an edge $\mathcal{W}(e)$ is the estimated frequency of the corresponding move instruction.*
2. E_r**:** *the move instructions that need to be added on control flow edges for which the two interval end points have different register assignments for the same compound interval. The weight of such an edge $\mathcal{W}(e)$ is the estimated frequency of the control-flow edge on which the move instruction is added.*

DEFINITION 5.3. **Assignment Optimization Problem:** *Given a set of basic intervals $b \in \mathcal{B}$ with $spilled(b) = false$,* CG $= \langle V, E = \{E_m \cup E_r\} \rangle$, IR, *and k uniform physical registers, find register assignment $reg(b)$ for every basic interval b such that the following objective function is minimized:*

$$\sum_{\forall e \in E, \quad e=(b_1,b_2) \, \wedge \, reg(b_1) != reg(b_2)} \mathcal{W}(e)$$

The assignment guides which additional register-to-register copy or exchange instructions need to be inserted in the IR.

Algorithm 6 presents a greedy heuristic to select a physical register for a basic interval b given the coalesce graph and the available set of physical register $avail$. $avail$ is updated as basic intervals expire. Map is a data structure that maps a physical register to a cost. Steps 3-7 find the physical registers and their associated costs that are already assigned to the neighbors of b in the coalesce graph (similar to the idea of biased coloring [Briggs et al. 1992]). Our approach takes into account the edges in E_r due to register-to-register moves. The greedy heuristics chooses a physical register $reg(b)$ with maximum cost, *i.e.,* the benefit of assigning the physical register to basic interval b.

```
function GetPreferredPhysical ()
    Input : A basic interval $b \in \mathcal{B}$, coalesce graph $G = \langle V, E = \{E_m \cup E_r\}\rangle$ and a set $avail$ currently available uniform physical registers
    Output: Find the assignment $reg(b)$
    $Map := \phi$;
    //Maximize the IR moves that can be removed
    for each edge $e = (b_1, b) \in E_m \cup E_r$ do
        if $b_1$ and $b$ do not intersect then
            $p := reg(b_1)$;
            if $p$ != null and $p \in avail$ then
                $Map(p) := Map(p) + \mathcal{W}(e)$;
    $ret$ := Find $p$ with maximum cost in $Map$;
    if $ret$ == null then
        $ret$ := Find any free physical register from $R$;
    Remove $ret$ from $avail$; $reg(b) := ret$; return $reg(b)$;
```

Figure 6. Greedy heuristic to choose a physical register that maximizes copy removal

THEOREM 5.4. *Register assignment using Algorithm 6 requires* $\mathcal{O}(|\mathcal{B}| + |\mathrm{IR}| + (|\mathcal{C}| * max_c))$ *space where* max_c *denotes the maximum number of basic intervals in a compound interval.*

Proof: The additional space requirement is due to the coalesce graph *CG* containing $|\mathcal{B}|$ number of nodes. E_m in the worst case ends up creating $|IR|$ edges. E_r adds edges between basic intervals of the same compound interval and hence needs $|\mathcal{C}| * max_c$ number of edges.□

THEOREM 5.5. *Register assignment using Algorithm 6 takes* $\mathcal{O}((|\mathcal{B}| * max_c) + |\mathrm{IR}| + (|\mathcal{E}| * (|\mathcal{C}| + |\mathcal{K}|^2)))$ *time.*

Proof: In addition to Theorem 5.3, before deciding a physical register for each basic interval b it is required to traverse each of the neighbors in *CG*. For all basic intervals, this adds over all $2 * |IR|$ time complexity for *IR* move instructions and $|\mathcal{B}| * max_c$ time complexity for E_r edges in *CG*.□

6. Allocation and Assignment with Register Classes

In the preceding sections, we have described register allocation and assignment for k physical registers that are uniform, *i.e.,* they are *independent* and *interchangeable* [Smith et al. 2004]. However, modern systems such as x86, HP RA-RISC, Sun SPARC, and MIPS come with physical registers which may not necessarily be interchangeable. For example, the Intel 32-bit x86 architecture provides eight integer physical registers, of which six are usable by Jikes RVM. These six physical registers are further divided into four high level overlapping *register classes* based on calling conventions and 8-bit operand accesses. Since the register classes may not necessarily be *disjoint*, a register allocator must take into account register classes during allocation and assignment to produce high quality machine code. In this section, we describe how allocation and assignment can be performed in the presence of register classes. We assume calling conventions related constraints are also expressed in additional register classes with infinite spill cost.

6.1 Constrained Allocation using BLG

Allocation in the presence of register classes can be achieved using the following two approaches:

1. Build *BLG* for each register class and apply Algorithm 3 to each *BLG* in a particular order starting with the most constrained register class that has fewer physical registers in a class. For example, in the 32-bit x86 architecture, we need to build four *BLG*s for four register classes in Jikes RVM and apply Algorithm 3 in the order *8 bit non-volatile* (EBX), *non-volatile* (EBX, EBP, and EDI), *8 bit volatile* (EAX, EBX, ECX, and EDX), and then for the complete integer register class. If a compound interval is spilled in a *BLG* for a register class, that decision need to be propagated to the other *BLG*s of other classes.
2. An alternative approach is to build a single *BLG*. During every visit of an interval end point in Algorithm 3, we make it unconstrained with respect to all register classes before another end point is visited. This approach is space-efficient as it builds only one *BLG* but can eagerly generate more spills than (1).

Our experimental results in Section 7 were obtained using Approach (1).

```
1  function ConstrainedAssignment ()
     Input  : Set of basic intervals $b \in \mathcal{B}$, $\forall b \in \mathcal{B}$ $regclass(b)$, a set of physical register classes $K$, a compile-time constant $num\_bucket$
     Output: Find the assignment $reg(b)$ and spill decision $spilled(b)$
     //Find total number of elements per $regclass$
2    for $b \in \mathcal{B}$ do
3        $cid$ := getClassId ($regclass(b)$);
4        $perClass[cid]++$;
     //Decide per bucket number of elements
5    for $i := 0; i < |K|; i++$ do
6        $perBucket[i]$ := $\lfloor perClass[i]/|K| \rfloor$ + 1;
7        $availBucket[i]$ := 0;
     //$assignOrder$ is a 2-d array of basic intervals;
     //Determine the bucket for $b$;
8    for $b \in \mathcal{B}$ in decreasing order of $SPILL(b)$ do
9        $cid$ := getClassId ($regclass(b)$);
10       $bucket$ := $availBucket[cid]$;
11       Append $b$ to $assignOrder[bucket][cid]$;
12       if $|assignOrder[bucket][cid]| > perBucket[cid]$ then
13           $availBucket[cid]++$;
     //Assign physical registers
14   for $i := 0; i < |K|; i++$ do
15       for $j := 0; j < num\_bucket; j++$ do
16           for $b \in assignOrder[i][j]$ do
17               findAssignment (b);
```

Figure 7. Bucket-based greedy heuristic to perform assignment in the presence of register classes.

6.2 Constrained Assignment and Move Coalescing

Given a coalesce graph (as defined in Section 5), when we try to find an assignment for a basic interval b, the register classes of the neighbors of b in the coalesce graph along with the register class of b, play a key role in selecting a physical register for b. An *IR* move instruction can be coalesced if source and destination basic intervals have a non-null intersection in their register classes.

Another key point in register assignment is that we no longer can rely on the increasing start point order for assignment of basic intervals since an early decision of physical register assignment of a register class may result in more symbolic registers being spilled later on or giving up other opportunities for coalescing. We define the register assignment problem in the presence of register classes as an optimization problem that may incur additional spills.

DEFINITION 6.1. **Constrained Assignment Optimization Problem:** *Given a set of basic intervals* $b \in \mathcal{B}$ *with* $spilled(b) = false$, $regclass(b)$ *indicating physical registers that can be assigned to each* b, CG $= \langle V, E = \{E_m \cup E_r\} \rangle$, *and* IR, *find a register assignment* $reg(b)$ *for a subset of basic intervals* $S \subseteq \mathcal{B}$ *such that the following objective function is minimized:*

$$\sum_{\forall b \in \mathcal{B} - S} SPILL(b) + \sum_{\forall e \in E,\quad e=(b_1,b_2)\ \wedge\ reg(b_1)!=reg(b_2)} \mathcal{W}(e)$$

Insert additional register-to-register copy or exchange instructions in the IR.

Algorithm 7 presents a bucket-based approach to register assignment that tries to strike a balance between register classes and spill cost. The $assignOrder$ data structure holds sorted basic intervals according to register classes in a two dimensional array. Each register class is represented as a unique integer id. Steps 2-4 compute the total number of basic intervals per register class. Steps 5-7 compute the number of elements per bucket. Steps 8-13 decide the appropriate bucket in $assignOrder$ where a basic interval should reside (based on next availability). Steps 14-17 find an assignment for basic intervals by traversing the $assignOrder$ array in a row major order. The heuristic for assigning a physical register to a basic interval follows a similar approach described in Section 5 except additional care must be taken to account for register class constraints. The details are provided in Algorithm 8.

```
function findAssignment ()
    Input  : A basic interval $b \in \mathcal{B}$, $\forall b \in \mathcal{B}$ $regclass(b)$, coalesce graph $G = \langle V, E = \{E_m \cup E_r\}\rangle$, a set of available physical registers $avail$
    Output: Find the assignment $reg(b)$
    Compute $Map$ using Steps 3-7 of Algorithm 6;
    $RMap$ := $Map$;
    for each edge $e = (b_1, b) \in E_m \cup E_r$ do
        if $b_1$ and $b$ intersect then
            for each $p$ in $Map$ do
                if $p$ can be assigned to $b_1$, i.e., $p \in regclass(b_1)$ then
                    $RMap(p)$ := $RMap(p)$ + $\mathcal{W}(e)$;
    $ret$ := Find $p$ with maximum cost in $RMap$;
    Follow Steps 7-11 of Algorithm 6;
```

Figure 8. Greedy heuristic to choose a physical register that maximizes copy removal in the presence of register classes

7. Experimental Results

We present an experimental evaluation of the *BLG* register allocation and assignment algorithms presented in this paper. The experimental setup consists of two compiler infrastructures, LLVM 2.7 [llv] and Jikes RVM 3.0.0 [jik]. The evaluations were performed on an Intel Xeon 2.4GHz system with 30GB of memory and running RedHat Linux (RHEL 5).

7.1 LLVM 2.7 (64-bit) evaluation

Benchmarks: We used ten benchmarks from the SPECCPU 2006 benchmark suite. The integer benchmarks used are `401.bzip2`, `429.mcf`, `458.sjeng`, `464.h264ref`, and `473.astar`. The floating-point benchmarks used are `410.bwaves`, `434.zeusmp`, `435. gromacs`, `444.namd`, and `470.lbm`. All the benchmarks were executed under the optimization level -O2 of LLVM. Since we invoked LLVM in static compilation mode, we ran each benchmark five times and reported the best of the 5 runs as the runtime performance measurement.

Comparison approaches: Experimental results are reported for the following cases:

1. `LLVMLS` – Baseline measurement using the default Linear Scan register allocator in LLVM; This allocator implements live-range splitting and differs from the standard linear scan algorithm [Poletto and Sarkar 1999] by introducing backtracking. These extensions are described in Wimmer et al. [Wimmer and Mössenböck 2005]. This algorithm also performs aggressive coalescing prior to register allocation.
2. `GC` – the Chaitin-Briggs [Chaitin et al. 1981, Briggs et al. 1994] register allocator. This implementation uses the same code base of Chaitin-Briggs allocator with aggressive coalescing that was used in [Cooper and Dasgupta 2006]. Details of the Chaitin-Briggs allocator can be found in [Briggs et al. 1994].
3. `BLG+LS` – the register allocation and assignment algorithm presented in Section 6 with the spill code generation algorithm from 1) above *i.e.,* after the allocation and assignment passes are completed using *BLG*, the *IR* is rewritten using the physical registers for the non-spilled variables and move code is inserted. The *IR* is then passed to the Linear Scan register allocator of LLVM to generate spill code)
4. `BLG+GS` – the register allocation and assignment algorithm presented in Section 6 with the spill code generation algorithm from 2) above *i.e.,* after allocation and assignment are completed using *BLG*, the *IR* is rewritten using the physical registers for the non-spilled variables and move code is inserted. The *IR* is then passed to the Chaitin-Briggs register allocator to generate spill code). For the *BLG* allocator, we set the compile-time constant num_bucket to 4.

Compile-time Comparison: Table 1 compares the compile-time overheads of *BLG* vs. *GC*. The measurements were obtained for functions with the largest interference graphs (in terms of number of nodes) in the SPECCPU 2006 benchmarks. Column 3 reports the total number of LLVM *IR* instructions for the max function. Column 4 and 5 report the total number of nodes and edges in the *IG* respectively. (We only report these numbers for the first iteration of the Chaitin-Briggs allocator – subsequent iterations require additional smaller interference graphs.) Column 5 and 6 report the total number of nodes and edges in *BLG* that only considers constrained interval end points (*i.e.,* those end points with MAXLIVE $> k$; unconstrained interval end points are not necessary, as described in Section 4). We define *Space Usage Ratio* metric as the ratio of the

Benchmark	max function	$\|IR\|$	*IG* #nodes	*IG* #edges	*BLG* #nodes	*BLG* #edges	Space Usage Ratio	*BLG* #nodes opt	*BLG* #edges opt
401.bzip2	sendMTFValues	3545	2693	53562	1844	9819	3.9	1721	8823
410.bwaves	bi_cgstab_block_	2083	1025	5430	134	269	3.4	134	269
429.mcf	read_min	440	279	3376	47	49	7.6	47	49
434.zeusmp	setup_	5147	3030	33138	387	1750	5.6	79	210
435.gromacs	do_inputrec	3519	1941	36606	64	142	11.3	39	67
444.namd	_ZN20ComputeNonbondedUtil30calc_self_energy_fullelect_fepEP9nonbonded	2244	907	6156	4	3	4.1	4	3
458.sjeng	std_eval	1316	812	7908	0	0	7.63	0	0
464.h264ref	SubPelBlockSearchBiPred	5787	4757	86092	356	921	13.7	53	55
470.lbm	LBM_handleInOutFlow	1162	643	5380	189	270	4.4	189	270
473.astar	_ZN6wayobj18makeobstaclebound2EPiiS0_	382	295	438	0	0	2.9	0	0

Table 1. Comparison of compile-time statistics between `BLG+LS` and `GC` for SPECCPU 2006 benchmarks. The number of compound intervals (*i.e.,* variables) for *BLG* is same as column 4. The *Space Usage Ratio* in column 8 is the ratio of the following two quantities: (1) sum of $|IR|$, *IG* nodes, and *IG* edges; (2) $|IR|$, *BLG* nodes, and *BLG* edges. Column 9 and 10 report the *BLG* nodes and edges after optimizing *BLG* for space.

Benchmark	`GC` (in sec)	`BLG+LS` (in sec)
401.bzip	938	10.9
410.bwaves	151.5	7.8
429.mcf	225.1	1.9
434.zeusmp	732.0	39.1
435.gromacs	9886.2	82.9
444.namd	1633.5	48.2
458.sjeng	1230.7	12.9
464.h264ref	5433.2	66.8
470.lbm	245.2	5.2
473.astar	1125.2	5.1

Table 2. Comparison of compile-times between `BLG+LS` and `GC` for SPECCPU 2006 benchmarks using LLVM static compiler. Clearly, `BLG+LS` achieves a significant reduction in compile-time relative to `GC`. The Interference Graph as a major compile-time bottleneck has also been observed in [Sarkar and Barik 2007, Cooper and Dasgupta 2006].

following two quantities: (1) sum of columns 3-5 ($|IG|$); (2) sum of columns 3, 6, and 7 ($|BLG|$). This metric varies from 2.9× to 13.7× in our case, indicating the lower space usage of *BLG* compared to *GC*. While theoretically both *IG* and *BLG* can be quadratic, in practice, we observe *BLG* to be much smaller than *IG*. We introduced an additional optimization for reducing the size of the *BLG* based on the simple observation that the constrained interval end points having the same set of variables live can be merged into a single merged interval end point. This optimization is applied on-the-fly as new interval end points are added to *BLG* using an efficient hashing mechanism for the end points. Column 9 and 10 report the total number of nodes and edges in *BLG* after applying the above optimization. The optimization reduces the *BLG* size for `401.bzip2`, `434.zeusmp`, `435.gromacs`, and `464.h264ref`.

Table 2 compares the total compilation time of each benchmark using *BLG* vs. *GC*. While compilation time can depend heavily on the algorithmic implementation, we observe a significant reduction in compile-time for *BLG* for all the benchmarks. Note that we did not implement `GC` ourselves but used it is as it is from [Cooper and Dasgupta 2006]. We expected *GC* to be slower than *BLG* but not by such large factors. One major source of compile-time inefficiency that we identified is the way register classes are handled in their implementation. Nonetheless, past work by Poletto and Sarkar [Poletto and Sarkar 1999] has shown a slowdown of factor 2 and also, Traub et al. [Traub et al. 1998] report a slowdown of factor 3.5 for large programs using *IG* vs. intervals. Further, *IG* being a compile-time bottleneck has also been observed in [Cooper and Dasgupta 2006, Sarkar and Barik 2007].

Runtime comparison: Figure 9 reports the relative performance improvement of the register allocation algorithm presented in this paper along with Chaitin-Briggs spill code generator, *BLG+GS*, compared to the original Chaitin-Briggs allocator, *i.e., GC*. We observe a performance improvement of up to 7.87% in `464.h264ref` benchmark and we do not observe any degradation in any of the benchmarks. While comparing our *BLG* allocator with Linear Scan spill code generator, *i.e., BLG+LS*, with that of LLVM's default register allocator *LLVMLS* (as shown in Table 3), we did not observe any noticeable performance difference. Note that the default LLVM register allocator uses live-range splitting and backtracking advanced techniques to

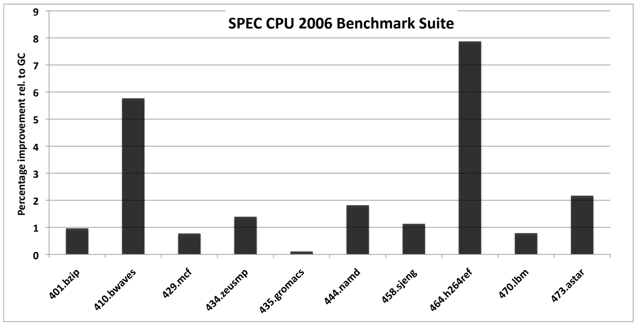


Figure 9. Percentage Improvement of execution times obtained by `BLG+GS`, (*i.e.,* BLG+Chaitin-Briggs spiller) compared to `GC` in the LLVM static compiler infrastructure for SPECCPU 2006 benchmarks.

Benchmark	`BLG+LS` execution time (in sec)	`LLVM+LS` execution time (in sec)
401.bzip	9.9	10.0
410.bwaves	2856.4	2853.1
429.mcf	6.7	6.8
434.zeusmp	40.4	40.5
435.gromacs	2079.1	2076.7
444.namd	38.1	38.1
458.sjeng	11.0	11.1
464.h264ref	1806.4	1806.4
470.lbm	1.6	1.6
473.astar	23.5	23.5

Table 3. Comparison of execution times obtained by `BLG+LS`, (*i.e.,* BLG+Linear Scan Spiller) compared to the default LLVM Linear Scan for SPEC CPU 2006 benchmarks using LLVM static compiler. Note that `LLVMLS` performs additional optimizations, such as live-range splitting and backtracking compared to `BLG+LS`.

help moderate register pressure during allocation and assignment. Live-range splitting for *BLG* can exploit the structure of the program as in [Lueh et al. 2000, Appel and George 2001] and is left for future work.

7.2 Jikes RVM 3.0.0 (32-bit) dynamic compiler evaluation

Benchmarks: We used the serial benchmarks in v2.0 of the Java Grande Forum (JGF) benchmark suite [Java Grande Forum] and Dacapo 2006 benchmark suite [Blackburn et al. 2006] to evaluate the performance of our register allocator. The JGF benchmarks consist of three sections. Section 1 contains microbenchmarks that are not relevant to a register allocation evaluation. Section 2 contains seven benchmarks (`Crypt`, `Heapsort`, `Sparsematmult`, `Sor`, `Series`, `LUFact`, and `FFT`) and Section 3 contains five large benchmarks (`Raytracer`, `Moldyn`, `Montecarlo`, `Euler`, and `Search`). For Dacapo benchmark suite, we report performance evaluation of nine benchmarks out of total eleven benchmarks. These include `antlr`, `bloat`, `fop`, `hsqldb`, `jython`, `luindex`, `pmd`, `xalan`, and `lusearch`.

Compiler: The boot image for Jikes RVM used a production configuration. Since the Jikes RVM release did not support generation of Intel exchange instruction, we modified its assembler to add this support. Jikes RVM uses `SSE` registers for storing double/floating point values. However, to the best of our knowledge, there does not exist a direct exchange instruction to swap values in `SSE` registers, so we generate three `xor` instructions to exchange a pair of float/double values. The exchange instructions are generated judiciously, *i.e.,* if there is a free physical register available for swapping the values, an exchange instruction is not generated [Boissinot et al. 2009]. For all Java runs, the execution times are reported for dynamic compilation (both runtime and compile-time) and use the methodology described in [Georges et al. 2007], *i.e.,* we report the average runtime performance of 30-runs within a single VM invocation along with the execution variance that uses a 95% confidence interval.[5]

Comparison approaches: Experimental results in Jikes RVM evaluation are reported for the following cases: 1) `LS` – Baseline measurement with Linear Scan register allocator in Jikes RVM that uses the algorithm from [Poletto and Sarkar 1999] with extensions for live-range "holes"; 2) `BLG` – the constrained register allocation algorithm presented in Section 6. The compile-time constant num_bucket in Figure 7 is set to 4 for all runs. Increasing this number to a higher value does not impact the runtime performance greatly.

---

[5] Due to lack of space, we omit all data for 95% confidence interval.

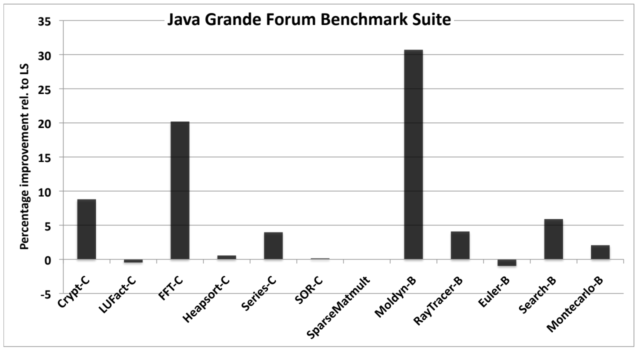


Figure 10. Percentage improvement of *BLG* compared to *LS* in Jikes RVM dynamic compiler for Java Grande

Runtime comparison: Figure 10 reports the relative performance improvements for all the benchmarks in JGF benchmark suite relative to the Linear Scan algorithm implemented in Jikes RVM 3.0.0. The *BLG* register allocator resulted in a performance improvement in the range of -0.1% to 30.7% (for `Moldyn`) in comparison with *LS*. For `Moldyn` benchmark, the absolute average execution time for *LS* is 43.9s with 95% confidence interval in the range 42.6s-43.7s. The absolute average execution time for *BLG* is 30.5s with 95% confidence interval in the range 30.4s-30.5s. For this benchmark, the most-frequently executed function is `force`. MAXLIVE for this function is >7. (Jikes RVM uses 8 SSE registers for storing double/float values, and one out of them, XMM7, is used for scratch register.) Spilling decisions for this method impact the performance of the benchmark significantly. *BLG* for this method coalesces more moves than *LS* and is able to spill 14 symbolic registers compared to 16 symbolic registers in *LS*. This is not surprising because *BLG* performs global spill decisions on the bipartite graph compared to the local decisions made by *LS* on active list.

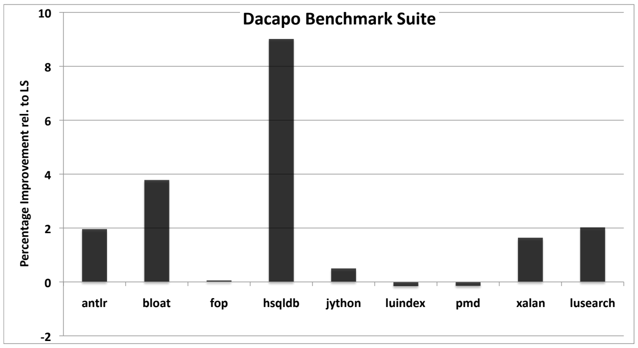


Figure 11. Percentage improvement of *BLG* compared to *LS* in Jikes RVM dynamic compiler for Dacapo

For Dacapo 2006 benchmark suite, we observe a performance improvement in the range of -0.2% to 9% (for `hsqldb`) for *BLG* register allocator compared to *LS* using the largest data set. The individual performance improvements are shown in Figure 11. For `hsqldb`, the absolute average execution time for *LS* is 8.5s with 95% confidence interval in the range 7.5s-10.5s. The absolute average execution time for *BLG* is 7.7s with 95% confidence interval in the range 7.5s-9.3s. Apart from `hsqldb`, we also observe performance improvements for `antlr`, `bloat`, and `lusearch`.

Compile-time: In a separate execution of all the Java Grande benchmarks to filter out the sole overhead of compile-time, we observe that our current Jikes RVM *BLG* implementation increases the overall compilation-time in the range of 4.1% to 18.5%. This modest increase in compile-time is acceptable given the runtime performance we achieve.

8. Related Work

Spill-free register allocation of general programs is NP-complete [Chaitin et al. 1981]. There exist a plethora of past works in using graph coloring-based approaches to spill-free register allocation [Chaitin et al. 1981, Briggs et al. 1994, Park and Moon 1998, George and Appel 1996]. The key data structures of a Graph Coloring based algorithm are live-ranges and the interference graph. One of the key limitations of graph coloring based register allocation is that the live-ranges introduce imprecision that may lead to making the interference graph uncolorable (like the one seen in Figure 3). In contrast, our approach builds on the simple foundations of Linear Scan register allocation like intervals and precisely captures liveness information using a novel *BLG* data structure, which is used for spill-free register allocation [Sarkar and Barik 2007].

Recently, the focus in graph coloring-based register allocation has shifted to SSA-based register allocation [Hack and Goos 2006, Brisk et al. 2005, Bouchez 2009, Pereira and Palsberg 2005, 2009, Braun et al. 2010]. In SSA representation, the interference graph is chordal and can be colored optimally in linear time. Like our approach and others in the literature [Appel and George 2001], current approaches to SSA register allocation separate between allocation and assignment phases in register allocation. However, an SSA register allocation requires the interference graph for allocation and assignment (thereby, incurs compile-time overhead) with an additional complexity of dealing with parallel-copy statements during out-of-ssa translation [Hack and Goos 2008]. Our *BLG* allocator does not need an interference graph for allocation and efficiently inserts a few register-to-register moves and exchange operations during assignment as opposed to expensive backtracking approaches to eliminate a large number of parallel-copy instructions in SSA-based register allocation.

Linear Scan [Poletto and Sarkar 1999, Traub et al. 1998, Wimmer and Mössenböck 2005, Thammanur and Pande 2004, Wimmer and Franz 2010] register allocation algorithms have been preferred for JIT-compilers such as Jikes [jik], HotSpot [Kotzmann et al. 2008], and LLVM [llv] due to their low compilation-time and space complexity. Compared to existing linear scan algorithms, our approach separates allocation and assignment phases. This leads to a much better global spilling decision using a novel bipartite graph. Traditional linear scan algorithms often combine allocation and assignment for efficiency reasons and hence end up making local spill decisions that lead to performance lag.

The graph coloring-based register allocation algorithm was first extended to handle register classes and aliasing by Smith et al [Smith et al. 2004]. The problem of spill-free register allocation is NP-complete even in the presence of register classes and aliasing [Lee et al. 2007]. The approach taken by Smith et al is to handle register classes and aliasing by exploiting the coloring constraints on each node of the interference graph. This approach is elegant and can be easily integrated into any graph coloring register allocation algorithm. More recently, a new approach based on puzzle solving was introduced by Pereira and Palsberg [Pereira and Palsberg 2008] to handle precoloring and aliasing issues in register allocation. Their approach views the register file as a puzzle and the program variables as puzzle pieces. For many common architectures, the register allocation using puzzles can be solved in polynomial time. Our *BLG* register allocator handles these architectural constraints without building the interference graph. For allocation phase, we construct *BLG* for each register class and propagate spill information across *BLG*'s of other register classes. For assignment phase, we use a bucket-based approach that strikes a balance between spill cost and move code optimization.

A bipartite graph-based register assignment phase was proposed by Zhang et al. [Zhang et al. 2004] that is performed on hot paths of already register allocated code, *i.e.,* as a post register allocation pass. The spilled variables on the hot path form one set of vertices of the bipartite graph where as the other set of vertices consists of the set of dead physical registers. An edge is added to their bipartite graph if both the spilled variable and dead physical register are alive in the same basic block. The weight of such an edge is the spill cost of the spilled variable in the basic block. Dead register assignment is then performed using weighted bipartite graph matching. This approach differs from our *BLG* allocator in many ways: 1) the nodes, edges, and weights of the bipartite graph are all different; 2) our bipartite liveness graph represents liveness information and solves the allocation phase of register allocation.

9. Conclusions

In this paper, we addressed the problem of developing a register allocation algorithm that builds on the simplicity of Linear Scan while improving its runtime performance. It does so by separating the allocation and assignment phases. The allocation phase is modeled as an optimization problem on Bipartite Liveness Graphs (*BLG*'s), a new data structure introduced in this paper. In the allocation and assignment phase, we focus on reducing the number of spill instructions by using register-to-register move and exchange instructions wherever possible to maximize the use of registers. We model register assignment as a second optimization problem that includes move coalescing, as well as register class constraints, and provide a heuristic solution to this problem as well. Compared to past work, our *BLG* register allocator incurs low compile-time overhead and results in high quality code. A prototype implementation of our BLG-based register allocation phase combined with the constrained assignment in Jikes RVM demonstrates runtime performances improvements in the range of -0.96% to 30.7% for Java Grande Forum and in the range of -0.16% to 9.013% for Dacapo benchmark suite. Additionally, we observe a performance improvement of up to 7.87% for SPECCPU 2006 benchmarks using our *BLG* register allocator that uses a graph coloring based spill code generator when compared to Chaitin-Briggs register allocator.

These results show that *BLG* register allocation algorithm is a promising alternate to the large body of register allocators existing today. Possible directions for future work include support for live-range splitting, and studying the impact of move and exchange instructions on code size compared to spill load/store instructions. Further, we would like to study the combined effect of *BLG* with instruction scheduling.

References

Jikes RVM. http://jikesrvm.org/.

The LLVM compiler infrastructure. http://llvm.org/.

Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 243–253, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: http://doi.acm.org/10.1145/378795.378854. URL `http://doi.acm.org/10.1145/378795.378854`.

S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press. doi: http://doi.acm.org/10.1145/1167473.1167488.

Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, and Christophe Guillon. Revisiting out-of-ssa translation for correctness, code quality and efficiency. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 114–125, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3576-0. doi: http://dx.doi.org/10.1109/CGO.2009.19. URL `http://dx.doi.org/10.1109/CGO.2009.19`.

Florent Bouchez. *A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases*. PhD thesis, April 2009.

Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *CGO '07*, pages 102–114, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2764-7. doi: http://dx.doi.org/10.1109/CGO.2007.26.

Matthias Braun, Christoph Mallon, and Sebastian Hack. Preference-Guided Register Assignment. In *Compiler Construction 2010*, volume 6011 of *Lecture Notes In Computer Science*, pages 205–223. Springer, 2010. ISBN 978-3-642-11969-9. doi: 10.1007/978-3-642-11970-5.

Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *PLDI '92*, volume 27, pages 311–321, New York, NY, 1992. ACM Press. ISBN 0-89791-475-9. URL `citeseer.ist.psu.edu/briggs92rematerialization.html`.

Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994. URL `citeseer.ist.psu.edu/briggs94improvements.html`.

P. Brisk, Dabiri F., Macbeth J., and Sarrafzadeh M. Polynomial time graph coloring register allocation. *14th International Workshop on Logic and Synthesis*, 2005.

G. J. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages 6*, pages 47–57, January 1981.

Keith D. Cooper and Anshuman Dasgupta. Tailoring graph-coloring register allocation for runtime compilation. In *CGO '06*, pages 39–49, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. doi: http://dx.doi.org/10.1109/CGO.2006.35.

Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996. URL `citeseer.ist.psu.edu/george96iterated.html`.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 57–76, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297033.

Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. In Shriram Krishnamurthi and Martin Odersky, editors, *Compiler Construction*, volume 4420 of *Lecture Notes in Computer Science*, pages 111–125. Springer Berlin / Heidelberg, 2007.

Sebastian Hack and Gerhard Goos. Optimal register allocation for SSA-form programs in polynomial time. *Inf. Process. Lett.*, 98(4): 150–155, 2006. ISSN 0020-0190. doi: http://dx.doi.org/10.1016/j.ipl.2006.01.008.

Sebastian Hack and Gerhard Goos. Copy coalescing by graph recoloring. In *PLDI '08*, pages 227–237, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: http://doi.acm.org/10.1145/1375581.1375610.

Lang Hames and Bernhard Scholz. Nearly optimal register allocation with pbqp. In David Lightfoot and Clemens Szyperski, editors, *Modular Programming Languages*, volume 4228 of *Lecture Notes in Computer Science*, pages 346–361. Springer Berlin / Heidelberg, 2006.

Java Grande Forum. The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.

Thomas Kotzmann, Christian Wimmer, Hansp eter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot$^{\mathrm{TM}}$client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):1–32, 2008. ISSN 1544-3566. doi: http://doi.acm.org/10.1145/1369396.1370017.

Jonathan Lee, K., Jens Palsberg, and Fernando Magno Pereira. Aliased register allocation for straight-line programs is NP-complete. In *Automata, Languages and Programming*, pages 680–691, 2007. doi: 10.1007/978-3-540-73420-8_59.

Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai. Fusion-based register allocation. *ACM Trans. Program. Lang. Syst.*, 22: 431–470, May 2000. ISSN 0164-0925.

Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In Jean-Luc Gaudiot, editor, *PACT '98*, pages 196–204, Paris, October 1998. IFIP,ACM,IEEE.

Fernando Magno Pereira and Jens Palsberg. SSA elimination after register allocation. In *CC '09*, pages 158–173, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00721-7.

Fernando Magno Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *APLAS'05*, pages 315–329, November 2005.

Fernando Magno Pereira and Jens Palsberg. Register allocation by puzzle solving. In *PLDI '08*, pages 216–226, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: http://doi.acm.org/10.1145/1375581.1375609.

Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5): 895–913, 1999. URL `citeseer.ist.psu.edu/poletto99linear.html`.

Vivek Sarkar and Rajkishore Barik. Extended Linear Scan: an Alternate Foundation for Global Register Allocation. In *CC '07*, 2007.

Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04*, pages 277–288, New York, NY, 2004. ACM. ISBN 1-58113-807-5. doi: http://doi.acm.org/10.1145/996841.996875.

Sathyanarayanan Thammanur and Santosh Pande. A fast, memory-efficient register allocation framework for embedded systems. *ACM Trans. Program. Lang. Syst.*, 26(6):938–974, 2004. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1034774.1034776.

Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *SIGPLAN PLDI'98*, pages 142–151, 1998. URL `citeseer.ist.psu.edu/article/traub98quality.html`.

Christian Wimmer and Michael Franz. Linear scan register allocation on SSA form. In *CGO '10*, pages 170–179, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-635-9. doi: http://doi.acm.org/10.1145/1772954.1772979.

Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *VEE '05*, pages 132–141, New York, NY, USA, 2005. ACM. ISBN 1-59593-047-7. doi: http://doi.acm.org/10.1145/1064979.1064998.

Kun Zhang, Tao Zhang, and Santosh Pande. Binary translation to improve energy efficiency through post-pass register re-allocation. In *EMSOFT '04*, pages 74–85, New York, NY, USA, 2004. ACM. ISBN 1-58113-860-1. doi: http://doi.acm.org/10.1145/1017753.1017769.